```python
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import cv2
!pip install opencv-python-headless
```

```python
import torch
import matplotlib.pyplot as plt
import torch.nn as nn
import seaborn as sns
from torch.utils.data import DataLoader, random_split
from PIL import Image
import torch.nn.functional as F
import torchvision.transforms as transforms
from torchvision.utils import make_grid
from torchvision.datasets import ImageFolder
from torchsummary import summary
from sklearn.metrics import confusion_matrix, classification_report
import itertools
from tqdm import tqdm
import os
```

```python
data_dir = "/kaggle/input/new-plant-diseases-dataset/New Plant Diseases Dataset(Augmented)/New Plant Dis
train_dir = data_dir + "/train"
valid_dir = data_dir + "/valid"
```

```python
classes = os.listdir(train_dir)
print(f"Classes found: {classes}")
print(f"Number of classes: {len(classes)}")

class_counts = {cls: len(os.listdir(os.path.join(train_dir, cls))) for cls in classes}
```

```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```python
transform = transforms.Compose([
    transforms.Resize((299, 299)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
# Because the dataset is already augmented we don't need to augment it manually.
```

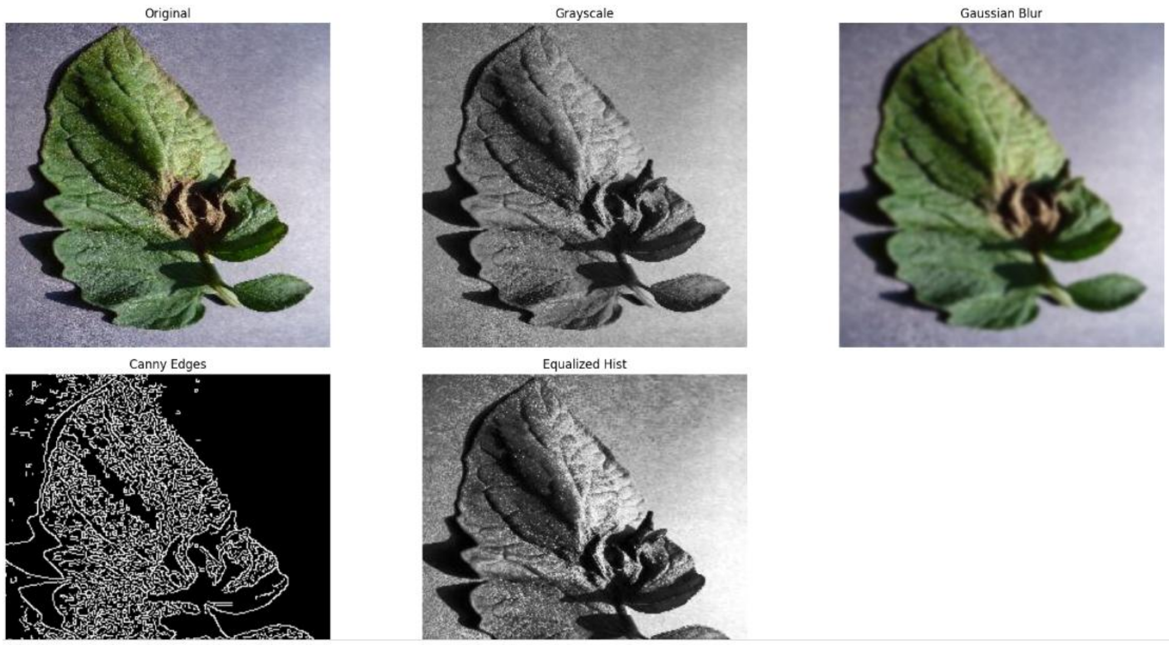
```python
full_dataset = ImageFolder(train_dir, transform=transform)
train_size = int(0.8 * len(full_dataset))
test_size = len(full_dataset) - train_size
train_dataset, test_dataset = random_split(full_dataset, [train_size, test_size])

valid_dataset = ImageFolder(valid_dir, transform=transform)
```

```python
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=2)
test_loader  = DataLoader(test_dataset, batch_size=32, shuffle=False, num_workers=2)
valid_loader = DataLoader(valid_dataset, batch_size=32, shuffle=False, num_workers=2)
```

```python
# Image Processing Visualization
def visualize_image_processing(img_path):
    img = cv2.imread(img_path)
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    gray = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
    blur = cv2.GaussianBlur(img_rgb, (7, 7), 0)
    edges = cv2.Canny(img_rgb, 100, 200)
    equalized = cv2.equalizeHist(gray)
    images = [img_rgb, gray, blur, edges, equalized]
    titles = ['Original', 'Grayscale', 'Gaussian Blur', 'Canny Edges', 'Equalized Hist']
    plt.figure(figsize=(18, 10))
    for i in range(5):
        plt.subplot(2, 3, i+1)
        if len(images[i].shape) == 2:
            plt.imshow(images[i], cmap='gray')
        else:
            plt.imshow(images[i])
        plt.title(titles[i])
        plt.axis('off')
    plt.tight_layout()
    plt.show()
```

```
sample_path = os.path.join(train_dir, classes[0], os.listdir(os.path.join(train_dir, classes[0]))[0])
visualize_image_processing(sample_path)
```



```
def train_model(model,train_loader,test_loader,criterion,optimizer,num_epochs=10):
    train_loss_history=[]
    val_loss_history=[]

    for epoch in range(num_epochs):
        print(f"Epoch {epoch+1}/{num_epochs}")
        model.train()
        running_loss=0.0
        correct,total=0,0

        for images,labels in tqdm(train_loader,desc="Training"):
            images,labels=images.to(device),labels.to(device)
            optimizer.zero_grad()
            outputs=model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss+=loss.item()
            _, predicted=torch.max(outputs,1)
            correct+=(predicted==labels).sum().item()
            total+=labels.size(0)

        train_loss=running_loss/len(train_loader)
        train_acc=correct/total
        train_loss_history.append(train_loss)
```

```python
        model.eval()
        val_loss = 0.0
        correct, total = 0, 0
        with torch.no_grad():
            for images, labels in tqdm(valid_loader, desc="Validation"):
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, predicted = torch.max(outputs, 1)
                correct += (predicted == labels).sum().item()
                total += labels.size(0)
        val_loss /= len(valid_loader)
        val_acc = correct / total
        val_loss_history.append(val_loss)

        print(f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f}")
        print(f"Val   Loss: {val_loss:.4f}, Val   Acc: {val_acc:.4f}\n")

    return model, train_loss_history, val_loss_history
```

```python
def test_model(model, test_loader, criterion, device):
    model.eval()
    test_loss = 0.0
    correct, total = 0, 0
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for images, labels in tqdm(test_loader, desc="Testing"):
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            test_loss += loss.item()

            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    test_loss /= len(test_loader)
    test_acc = correct / total

    # Compute confusion matrix
    conf_matrix = confusion_matrix(all_labels, all_preds)

    print(f"Test Loss: {test_loss:.4f}, Test Acc: {test_acc:.4f}")
    print("Confusion Matrix:")
    print(conf_matrix)

    return test_loss, test_acc, conf_matrix
```

```
import torchvision.models as models
import torch.optim as optim

num_classes = len(full_dataset.classes)
model = models.inception_v3(pretrained=False, aux_logits=False)
model.fc = nn.Linear(model.fc.in_features, num_classes)
model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
model, train_loss_history, val_loss_history = train_model(
    model, train_loader, valid_loader, criterion, optimizer, num_epochs=12
)
```

```
Epoch 1/12
Training: 100%|██████████| 1758/1758 [13:43<00:00,  2.14it/s]
Validation: 100%|██████████| 550/550 [02:21<00:00,  3.90it/s]
Train Loss: 1.0101, Train Acc: 0.6962
Val   Loss: 0.3937, Val   Acc: 0.8722

Epoch 2/12
Training: 100%|██████████| 1758/1758 [13:46<00:00,  2.13it/s]
Validation: 100%|██████████| 550/550 [01:21<00:00,  6.79it/s]
Train Loss: 0.3557, Train Acc: 0.8861
Val   Loss: 0.3329, Val   Acc: 0.8897

Epoch 3/12
Training: 100%|██████████| 1758/1758 [13:46<00:00,  2.13it/s]
Validation: 100%|██████████| 550/550 [01:21<00:00,  6.79it/s]
Train Loss: 0.2265, Train Acc: 0.9269
Val   Loss: 0.1729, Val   Acc: 0.9472

Epoch 4/12
Training: 100%|██████████| 1758/1758 [13:46<00:00,  2.13it/s]
Validation: 100%|██████████| 550/550 [01:20<00:00,  6.79it/s]
Train Loss: 0.1785, Train Acc: 0.9428
Val   Loss: 0.1800, Val   Acc: 0.9470
```

[24]:
```
test_loss, test_acc, conf_matrix = test_model(model, test_loader, criterion, device)

class_labels = [str(i) for i in range(len(conf_matrix))]

plt.figure(figsize=(16, 12))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues",
            xticklabels=class_labels, yticklabels=class_labels,
            linewidths=1, linecolor="gray", square=False,
            annot_kws={"size": 10})

# Adjust label font sizes
plt.xlabel("Predicted Labels", fontsize=12, labelpad=15)
plt.ylabel("True Labels", fontsize=12, labelpad=15)
plt.title("Confusion Matrix", fontsize=14, pad=20)

# Rotate labels to prevent overlap
plt.xticks(fontsize=10, rotation=90)  # Rotate for better visibility
plt.yticks(fontsize=10, rotation=0)

plt.tight_layout()
plt.show()
```

```
Testing: 100%|██████████| 440/440 [01:48<00:00,  4.05it/s]
Test Loss: 0.0475, Test Acc: 0.9858
```

[25]:
```
torch.save(model.state_dict(), "inceptionV3_new_plant_disease_from_scratch.pth")
print("Model saved as inceptionV3_new_plant_disease_from_scratch.pth")
```

```
Model saved as inceptionV3_new_plant_disease_from_scratch.pth
```

[26]:
```
import matplotlib.pyplot as plt

def plot_loss_curve(train_loss, val_loss):
    epochs = range(1, len(train_loss) + 1)

    plt.figure(figsize=(8, 6))
    plt.plot(epochs, train_loss, label="Training Loss", marker="o", linestyle="-")
    plt.plot(epochs, val_loss, label="Validation Loss", marker="o", linestyle="-")

    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.title("Training vs Validation Loss")
    plt.legend()
    plt.grid(True)
    plt.show()

plot_loss_curve(train_loss_history, val_loss_history)
```

```python
import numpy as np
import matplotlib.pyplot as plt
from skimage import exposure

class ImageProcessor:
    @staticmethod
    def clahe(img):
        lab = cv2.cvtColor(img, cv2.COLOR_BGR2LAB)
        l, a, b = cv2.split(lab)
        clahe = cv2.createCLAHE(clipLimit=3.0, tileGridSize=(8,8))
        cl = clahe.apply(l)
        limg = cv2.merge((cl,a,b))
        return cv2.cvtColor(limg, cv2.COLOR_LAB2BGR)

    @staticmethod
    def gaussian_denoise(img):
        return cv2.GaussianBlur(img, (5,5), 0)

    @staticmethod
    def sharpen(img):
        kernel = np.array([[-1,-1,-1], [-1,9,-1], [-1,-1,-1]])
        return cv2.filter2D(img, -1, kernel)

    @staticmethod
    def hist_equalization(img):
        img_yuv = cv2.cvtColor(img, cv2.COLOR_BGR2YUV)
        img_yuv[:,:,0] = cv2.equalizeHist(img_yuv[:,:,0])
        return cv2.cvtColor(img_yuv, cv2.COLOR_YUV2BGR)
```

```python
    def adaptive_threshold(img):
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        return cv2.adaptiveThreshold(gray, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                     cv2.THRESH_BINARY, 11, 2)

    @staticmethod
    def hybrid_1(img):
        processed = ImageProcessor.clahe(img)
        processed = ImageProcessor.hist_equalization(processed)
        return ImageProcessor.sharpen(processed)

# %% [code]
# Visualization Function
def visualize_processing(original, processed, title):
    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.imshow(cv2.cvtColor(original, cv2.COLOR_BGR2RGB))
    plt.title("Original")
    plt.axis('off')

    plt.subplot(1, 2, 2)
    if len(processed.shape) == 2:  # Grayscale image
        plt.imshow(processed, cmap='gray')
    else:  # Color image
        plt.imshow(cv2.cvtColor(processed, cv2.COLOR_BGR2RGB))
    plt.title(title)
    plt.axis('off')
    plt.show()
```

```
        plt.imshow(processed, cmap='gray')
    else:  # Color image
        plt.imshow(cv2.cvtColor(processed, cv2.COLOR_BGR2RGB))
    plt.title(title)
    plt.axis('off')
    plt.show()

# %% [code]
# Get a sample image
sample_path = next((input_dir / "train").rglob("*.JPG"))
sample_img = cv2.imread(str(sample_path))

# Define processing techniques to visualize
techniques = [
    ("CLAHE", ImageProcessor.clahe),
    ("Gaussian Denoising", ImageProcessor.gaussian_denoise),
    ("Sharpening", ImageProcessor.sharpen),
    ("Histogram Equalization", ImageProcessor.hist_equalization),
    ("Adaptive Threshold", ImageProcessor.adaptive_threshold),
    ("Hybrid (CLAHE+Hist+Sharp)", ImageProcessor.hybrid_1)
]

# Visualize each technique
for name, processor in techniques:
    processed_img = processor(sample_img.copy())
    visualize_processing(sample_img, processed_img, name)
```

```
!pip install efficientnet_pytorch torchvision

# %% [code]
# Imports
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import transforms, datasets
from torch.utils.data import DataLoader
from efficientnet_pytorch import EfficientNet
from tqdm import tqdm

# Set device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")

# %% [code]
# Dataset paths
data_dir = "/kaggle/input/new-plant-diseases-dataset/New Plant Diseases Dataset(Augmented)/New Plan
train_dir = os.path.join(data_dir, "train")
valid_dir = os.path.join(data_dir, "valid")

# Get class names
classes = sorted(os.listdir(train_dir))
print(f"Classes found: {len(classes)}")
```

```
transform = transforms.Compose([
    transforms.Resize((300, 300)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

# Datasets
train_dataset = datasets.ImageFolder(train_dir, transform=transform)
valid_dataset = datasets.ImageFolder(valid_dir, transform=transform)

# Dataloaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=2)
valid_loader = DataLoader(valid_dataset, batch_size=32, shuffle=False, num_workers=2)

# %% [code]
# EfficientNet-B3 Model
model = EfficientNet.from_pretrained('efficientnet-b3', num_classes=len(classes))
model = model.to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# %% [code]
# Training function
def train_model(model, train_loader, valid_loader, criterion, optimizer, epochs=5):
    best_acc = 0.0
```

```
    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        # Training loop
        for images, labels in tqdm(train_loader, desc=f"Epoch {epoch+1}/{epochs}"):
            images, labels = images.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        train_acc = 100 * correct / total
        train_loss = running_loss / len(train_loader)

        # Validation
        val_acc, val_loss = validate(model, valid_loader, criterion)

        # Save best model
```

```python
def validate(model, loader, criterion):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for images, labels in loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)

            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return 100 * correct / total, running_loss / len(loader)

# %% [code]
# Train the model
train_model(model, train_loader, valid_loader, criterion, optimizer, epochs=5)

# %% [code]
# Load best model and test
model.load_state_dict(torch.load('best_model.pth'))
val_acc, val_loss = validate(model, valid_loader, criterion)
print(f"\nFinal Validation Accuracy: {val_acc:.2f}%")
```